7P

Artificial Intelligence Project---RLE and MIT Computation Center

Memo 12

PROGRAMS IN LISP

by John McCarthy

1. Introduction

This memo depends only on the RLE QPR No. 53 discussion of LISP. Its objective is to add to the system of that report a program feature. This takes the form of allowing functions to be defined by programs including sequences of Fortran-like statements, eg.

y=cons[ff[subst[A;y;z]];(A,B)]

Such a feature was included in the informal version of LISP from which we hand-compiled into SAP and is also available in the latest version of the apply operator. The version in the present apply operator is added merely as a convenience and does not have the mathematical elegance that we require. In the present memorandum, I will try to add a program feature to the system in a systematic way. It may be some time before this version is available in the programming system.

Since all computable functions can be expressed in LISP without the program feature, this feature can only be regarded as a convenience. However, it is a convenience at which we cannot afford to sneer.

This convenience has two aspects.

1. It allows many programs tc be written more concisely and with a greater independence of the parts than does the straight recursive function notation.

2. It expresses the fact that many recursive functions are simpler than recursive functions are in general and can be compiled in a special way into programs that are more economical in time and data storage space than the most general form of recursive function. Specifically, this form of program indicates that saving is unnecessary in certain cases and also that certain data are obsolete at certain points in the program and that therefore it is possible to change old list structure rather than the prepare a new list structure with the changes.

We shall discuss the two virtues of the program feature separately.

2. How Programs Shall Be Written

In section 7 of the QPR article we describe a way of interpreting a program as a function. The state of the computer is represented by a vector ξ whose components are the variables of the program. Each section of the program is regarded as a function f which gives the change in ξ that takes place when this section executed, i.e. ξ is changed to $\xi' = f(\xi)$.

Consider a single replacement statement e.g.

y=cons[ff[subst[A;y;z]];(A,B)]

The variables y and z are components of the quantity ξ, and we want to consider the function f which describes the transformation $\xi' = f(\xi)$ which occurs when the above replacement statement is executed. We can do this as follows:

We regard ξ as a list of the form
((variable,value),(variable,value),etc.)
and we define a function change as follows:

change[vec;var;val] = [null[vec→ list[list[var;val]];caar[vec] = var→cons[list[var;val];cdr[vec]];T→cons[car[vec];change[cdr[vec];var;val]]]]

Then we can write

f[ξ] = change[ξ;Y; cons[ff[subst[A;assoc[Y;ξ];assoc[Z;ξ]]];(A,B)]]

The effect of executing in order the replacement statements represented by the functions f,g and h is described by the function $\lambda[[\xi];h[g[f[\xi]]]]$. The reversal of order is caused by the fact that the composition of functions is described by listing the functions in an order opposite to their order of performance.

We can now give an S-function which gives the function corresponding to a sequence of statements. A statement is represented by a pair consisting of the variable to be changed and the expression which defines its new value. Thus the statement

y = cons[ff[subst[A;y;z]];(A,B)]

is represented by the pair

(Y,(CONS,(FF,(SUBST,(QUOTE,A),Z)),(QUOTE,(A,B))))

and the program is represented by the list of its statements in the order in which they are to be executed.

We now define the function

program[π;ξ] = [null[π]→ξ;T→program[cdr[π];change[ξ;caar[π];eval[cadar[π];ξ]]]]

[illegible] is a program in the form given above and ξ is a state

This form of program feature has the advantage that the vector ξ can be different on different uses of a program and different ξs with the same variables can have different values assigned to the variables. In particular, if a program is started with NIL for ξ all assignments start from scratch.

The simple form of program feature given above can profitably be elaborated. The following features may be added.

1. Transfers of control. We can do this by making IL (for instruction location) a special variable. The program function then becomes

$$program2[\pi;\xi]=[null[\pi]\rightarrow\xi;caar[\pi]=IL\rightarrow program2[eval[cadar[\pi];\xi];\xi];T\rightarrow program2[cdr[\pi];eval[cadar[\pi];\xi]]]$$

This allows transfer of control to any part of the program that can be computed. However, without a system of labelling statements it is difficult to describe these computations. One way of labelling statements is to describe the individual program step by a triplet rather than by a pair. This would require a slight modification of the above function.

Another method is to keep a separate list of pairs whose elements consist of labels and the locations of the corresponding statements.

2. Simultaneous change of several variables. As an example of this feature, suppose we wish tc exchange the values of the variables X and Y. The program ((X,Y),(Y,X)) will not do it because X is changed before Y is computed. If we write (U,Y),(X,Y),(Y,V)) we get a program for this, but it is objectionable to have to introduce the auxiliary variable U.

Let us consider a new form of program each of whose statements generates a list of pairs each element of which consists of a variable and a value. Our rule is to change the vector ξ according to the list of pairs produced. We have

$$program3[\pi;\xi]=[null[\pi]\rightarrow\xi;T\rightarrow program3[cdr[\pi];change2[\xi;eval[car[\pi];\xi]]]]$$

The function change2 takes the list of pairs produced by the program step and makes the indicated changes in the vector ξ.

I shall write change2 assuming that there is an ordering of atomic symbols denoted by $<$ and that the vector ξ and the vectors produced by the program steps have their first elements of the pairs

ordered ascendingly.

We have

change2[vec;list] = [null[vec]→list;
null[list]→vec;caar[vec]=caar[list]→
cons[car[list];change[cdr[vec];cdr[list]];
caar[vec]<caar[list]→cons
[car[vec];change[cdr[vec];list]];T→
cons[car[list];change[vec;cdr[list]]]]

It is not difficult to combine the previous incorporation of transfer instructions with the present inclusion of simultaneous changes.

3. Efficiency of Programs

In developing LISP our first goal is to describe a language which is as powerful as possible from the point of view of the programmer. More precisely, we wish to be able to describe the transformation between the input of a program and the desired output as directly as possible. How is it possible to say that one programming language is more powerful than another when it is known that a programming language containing only the 704 instructions SUB, STO, and TMI is adequate for describing any computable function? We must be able to refine the concept of power of a language for describing computations beyond merely taking into account the set of processes that can be described in order to usefully compare programming languages.

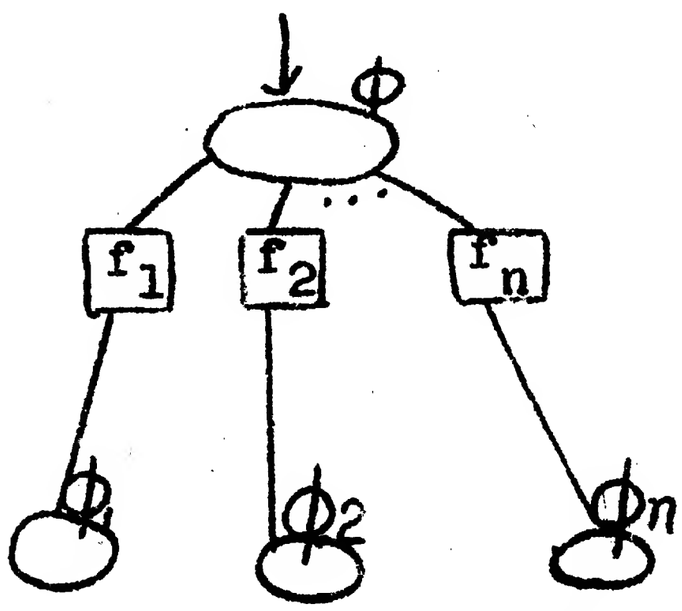
One approach to refining the concept has to do with auxiliary quantities occurring in the calculation; I shall give two examples. First, since LISP operates on the 704, the 704 translation of a LISP program describes the same process. However, the 704 program makes explicit reference to auxiliary quantities and processes such as the free storage list, the public push-down list, the contents of index registers, and many other registers and subroutines that are hidden from the LISP programmer. The superiority of LISP to machine language from the point of the view of the programmer is that he need not introduce these auxiliary entities in order to describe his process. As a second example consider matrix multiplication. The machine operations which occur in carrying it out include n^3 multiplications and the $(n-1)n^2$ additions which go into the evaluation of the elements of the product matrix. An actual

machine program for matrix multiplication includes other

arithmetic operations, for example additions to the subscripts of matrix elements, but the purpose of these is merely to insure that the correct matrix elements are multiplied and added and the results put in the right place. We shall consider one programming system more powerful than another if functions which can be described directly in the one require the description of auxiliary computation in the other.

Of course, when the computations are to be performed on an actual computer, the auxiliary processes still have to be performed whether the programmer has to describe them or whether they are automatically generated by the translation of the programming language into machine instructions.

It was shown in the QPR extract that a program described by a flow chart could have this description readily translated into a description using recursive functions. The converse is not true. A recursive function such as subst can be described by a computer program only if the use of the public push-down list is explicitly incorporated in the program. In fact, flow charts translate into recursive functions with special properties. Recall that a portion of a flow chart as shown in the figure gives rise to a recursive function



$$\phi[x;y]=[p_1[x;y]\rightarrow\phi_1[f_1[x;y]];\ldots;p_n[x;y]\rightarrow\phi_n[f_n[x;y]]]$$

The computation of any of the ϕ's is referred after a decision is made (by the p's) and a preliminary calculation is made (by an f) to the evaluation of a single ϕ_1. Contrast this with the fact that the evaluation of subst[x;y;z] generally involves the calculation of both subst[x;y;car[z]] and subst[x;y;cdr[z]].

This implies that

1. The program for a function ϕ does not use the programs for the other ϕ's as subroutines but transfers control to them irrevocably. (In 704 language, a TRA is used rather than a TSX). Therefore, no push-down list is required to save temporary results.

2. The vector $\mathfrak{x}$ can be regarded as a single-valued quantity which changes during the execution of the program. In the more general kind of recursion functions of the form $f[\mathfrak{x};g[\mathfrak{x}]]$ may occur which means that the original value of $\mathfrak{x}$ must be preserved while $g[\mathfrak{x}]$ is computed. The LISP program feature described in the previous section allow programs to be expressed that require this kind of saving.

We may ask how the machine language program corresponding to a LISP program looks. The answer is that the vector $\mathfrak{x}$ in the machine language program contains many more components than the original LISP program including components describing the push-down situation and the recursion situation.

The generality of the recursive function way of describing computation processes has certain disadvantages which have to be overcome. If we describe a computation process as a recursive function which is of the special kind that can be represented by a flow chart and our compilation process (either by a compiler or by hand) does not take this into account an inefficient program will result. The following inefficiencies can occur.

1. In general when a block of computation is performed a complete new vector $\mathfrak{x}$ must be computed and the old one saved. (If the old one is never used it will be abandoned and later picked up by the reclamation program). In general, some components will be recomputed and the others will be copied along the top line. In the special case, however, it would suffice to compute new values for some of the components and store the result in the vector. Such a process cannot be accomplished using car, cdr and cons since the functions never change existing list structure.

2. When control is transfered without the possibility of return the original state of the computation need not be saved. The "transfer" instruction in the program feature of the previous section has this efficiency.

3. If only one vector ξ is required another efficiency is possible. In the present system in order to find the value of a variable it is necessary to search the vector ξ. If there is but one such vector and the program is compiled the instructions can be written so as to refer directly to the component, and no search is required.

We hope to explore the possibility of including in the compiler ways of determining when a special form of calculation can be used with a saving of time and space. An alternative is to provide a way for the programmer to indicate that a calculation has special properties.

CS-TR Scanning Project
Document Control Form

Date : 11/30/95

Report # AIM-12

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)

☐ Other:_____

Document Information

Number of pages: 7(11-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:

☒ Single-sided or

☐ Double-sided

Intended to be printed as :

☒ Single-sided or

☐ Double-sided

Print type:

☐ Typewriter ☐ Offset Press ☐ Laser Print

☐ InkJet Printer ☐ Unknown ☒ Other: MIMEOGRAPH

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☐ Cover Page

☐ Spine ☐ Printers Notes ☐ Photo negatives

☐ Other: _____

Page Data:

Blank Pages(by page number): _____

Photographs/Tonal Material (by page number): _____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-7)	1-7
(8-11) SCANCONTROL TRGT'S (3)	

Scanning Agent Signoff:

Date Received: 11/30/95 Date Scanned: 12/14/95 Date Returned: 12/14/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**